

Towards “Propagation = Logic + Control”

Sebastian Brand¹ and Roland H.C. Yap²

¹ National ICT Australia, Victoria Research Lab, Melbourne, Australia

² School of Computing, National University of Singapore, Singapore

Abstract. Constraint propagation algorithms implement logical inference. For efficiency, it is essential to control whether and in what order basic inference steps are taken. We provide a high-level framework that clearly differentiates between information needed for controlling propagation versus that needed for the logical semantics of complex constraints composed from primitive ones. We argue for the appropriateness of our *controlled propagation* framework by showing that it captures the underlying principles of manually designed propagation algorithms, such as literal watching for unit clause propagation and the lexicographic ordering constraint. We provide an implementation and benchmark results that demonstrate the practicality and efficiency of our framework.

1 Introduction

Constraint programming solves combinatorial problems by combining search and logical inference. The latter, constraint propagation, aims at reducing the search space. Its applicability and usefulness relies on the availability of efficiently executable propagation algorithms.

It is well understood how primitive constraints, e.g. indexical constraints, and also their reified versions, are best propagated. We also call such primitive constraints *pre-defined*, because efficient, special-purpose propagation algorithms exist for them and many constraint solving systems provide implementations. However, when modelling problems, one often wants to make use of more complex constraints whose semantics can best be described as a combination of pre-defined constraints using logical operators (i.e. conjunction, disjunction, negation). Examples are constraints for breaking symmetries [FHK⁺02] and channelling constraints [CCLW99].

Complex constraints are beneficial in two aspects. Firstly, from a reasoning perspective, complex constraints give a more direct and understandable high-level problem model. Secondly, from a propagation perspective, the more global scope of such constraints can allow stronger inference. While elaborate special-purpose propagation algorithms are known for many specific complex constraints (the classic example is the `alldifferent` constraint discussed in [Rég94]), the diversity of combinatorial problems tackled with constraint programming in practice implies that more diverse and rich constraint languages are needed.

Complex constraints which are defined by logical combinations of primitive constraints can always be decomposed into their primitive constituents and

Boolean constraints, for which propagation methods exist. However, decomposing in this way may

- (A) cause redundant propagation, as well as
- (B) limit possible propagation.

This is due to the loss of a global view: information between the constituents of a decomposition is only exchanged via shared constrained variables.

As an example, consider the implication constraint $x = 5 \rightarrow y \neq 8$ during constructive search. First, once the domain of x does not contain 5 any more, the conclusion $y \neq 8$ is irrelevant for the remainder of the search. Second, only an instantiation of y is relevant as non-instantiating reductions of the domain of y do not allow any conclusions on x . These properties are lost if the implication is decomposed into the reified constraints $(x = 5) \equiv b_1$, $(y \neq 8) \equiv b_2$ and the Boolean constraints $\text{not}(b_1, b'_1)$, $\text{or}(b'_1, b_2)$.

Our focus is point (A). We show how *shared control information* allows a constraint to signal others what sort of information is relevant to its propagation or that any future propagation on their part has become irrelevant to it. We address (B) to an extent by considering implied constraints in the decomposition. Such constraints may be logically redundant but not operationally so. Control flags connecting them to their respective antecedents allow us to keep track of the special status of implied constraint, so as to avoid redundant propagation steps. Our proposed control framework is naturally applicable not only to the usual tree-structure decomposition but also to those with a more complex DAG structure, which permits stronger propagation.

Our objective is to capture the *essence* of manually designed propagation algorithms, which implicitly merge the separate aspects of logic and control. We summarise this by *Propagation = Logic + Control* in the spirit of [Kow79]. The ultimate goal of our approach is a fully automated treatment of arbitrary complex constraints specified in a logic-based constraint definition language. We envisage such a language to be analogous to CLP but focused on propagation. Our framework would allow users lacking the expertise in or the time for the development of specialised propagation to rapidly prototype and refine propagation algorithms for complex constraints.

Preliminaries

Consider a finite sequence of different variables $X = x_1, \dots, x_m$ with respective domains $D(x_1), \dots, D(x_m)$. A **constraint** C on X is a pair $\langle S, X \rangle$. The set S is an m -ary relation and a subset of the Cartesian product of the domains, that is, $S \subseteq D(x_1) \times \dots \times D(x_m)$. The elements of S are the **solutions** of the constraint, and m is its *arity*. We assume $m \geq 1$. We sometimes write $C(X)$ for the constraint and often identify C with S .

We distinguish pre-defined, **primitive** constraints, such as $x = y$, $x \leq y$, and **complex** constraints, constructed from the primitive constraints and the logical operators \vee, \wedge, \neg etc. For each logical operator there is a corresponding

Boolean constraint. For example, the satisfying assignments of $x \vee y = z$ are the solutions of the constraint $\text{or}(x, y, z)$. The **reified** version of a constraint $C(X)$ is a constraint on X and an additional Boolean variable b reflecting the truth of $C(X)$; we write it as $C(X) \equiv b$. Complex constraints can be **decomposed** into a set of reified primitive constraints and Boolean constraints, whereby new Boolean variables are introduced. For example, the first step in decomposing $C_1 \vee C_2$ may result in the three constraints $C_1 \equiv b_1$, $C_2 \equiv b_2$, and $\text{or}(b_1, b_2, 1)$.

Constraint **propagation** aims at inferring new constraints from given constraints. In its most common form, a single constraint is considered, and the domains of its variables are reduced without eliminating any solution of the constraint. If every domain is maximally reduced and none is empty, the constraint is said to be **domain-consistent** (DC). For instance, $x < y$ with $D(x) = \{1, 2\}$, $D(y) = \{1, 2, 3\}$ can be made domain-consistent by inferring the constraint $y \neq 1$, leading to the smaller domain $D(y) = \{2, 3\}$.

Decomposing a complex constraint may hinder propagation. For example, DC-establishing propagation is guaranteed to result in the same domain reductions on a constraint and its decomposition only if the constraint graph of the decomposition is a tree [Fre82]. For instance, the constraints of the decomposition of the constraint $(x > y) \wedge (x < y)$ considered in isolation do not indicate its inconsistency.

2 Logic and Control Information

A complex constraint expressed as a logical combination of primitive constraints can be decomposed into its primitive parts. However, such a naive decomposition has the disadvantage that it assigns equal relevance to every constraint. This may cause redundant reasoning to take place for the individual primitive constraints and connecting Boolean constraints. We prevent this by maintaining fine-grained control information on whether the *truth* or *falsity* of individual constraints matters. We say that a truth status of a constraint is **relevant** if it entails the truth status of some other constraint.

We focus on the disjunction operator first.

Proposition 1. *Suppose C is the disjunctive constraint $C_1 \vee C_2$. Consider the truth status of C in terms of the respective truth statuses of the individual constraints C_1, C_2 .*

- *If the falsity of C is asserted then the falsity of C_1 and C_2 can be asserted.*
- *If the truth of C is asserted then the falsity of C_1 and C_2 is relevant, but not their truth.*
- *If the truth of C is queried then the truth of C_1 and C_2 is relevant, but not their falsity.*
- *If the falsity of C is queried then the falsity of only one of C_1 or C_2 is relevant, but not the their truth.*

Proof. Let the reified version of C be $(C_1 \vee C_2) \equiv b$ and its partial decomposition be $C_1 \equiv b_1$, $C_2 \equiv b_2$, $\text{or}(b_1, b_2, b)$. The following cases can occur when asserting or querying C .

Case $b = 0$. Then C_1 and C_2 must both be asserted to be false.

Case $b = 1$.

- Suppose C_1 is found to be true. This means that both the truth and the falsity of C_2 , hence C_2 itself, have become irrelevant for the remainder of the current search. Although this simplifies the representation of C to C_1 , it does not lead to any inference on it. In this sense, the truth of C_1 is useless information.

The case of C_2 being true is analogous.

- Suppose C_1 is found to be false. This is useful information as we now must assert the truth of C_2 , which may cause further inference in C_2 .

The case of C_2 being false is analogous.

Only falsity of C_1 or C_2 is information that may cause propagation. Their truth is irrelevant in this respect.

Case b unknown. We now assume that we know what aspect of the truth status of C is relevant: its truth or its falsity. If neither is relevant then we need not consider C , i. e. C_1 and C_2 , at all. If both the truth and falsity of C are relevant, the union of the individual cases applies.

Truth of C is queried:

- Suppose C_1 or C_2 is found to be true. This means that C is true, and knowing either case is therefore useful information.
- Suppose C_1 is found to be false. Then the truth of C depends on the truth of C_2 . The reasoning for C_2 being false is analogous.

The truth of both C_1 and C_2 matters, but not their falsity.

Falsity of C is queried:

- Suppose C_1 or C_2 is found to be true. While this means that C is true, this is not relevant since its falsity is queried.
- Suppose C_1 is found to be false. Then the falsity of C depends on the falsity of C_2 . Now suppose otherwise that C_1 is queried for falsity but *not* found to be false. If C_1 is not false then C cannot be false. It is important to realise that this reasoning is independent of C_2 . The reasoning for C_2 being false is symmetric.

In summary, to determine the falsity of C , it suffices to query the falsity of *just one* of C_1 or C_2 . □

Fig. 1 shows the flow of control information through a disjunction. There, and throughout the rest of this paper, we denote a truth query by *chk-true* and a falsity query by *chk-false*.

Analogous studies on control flow can be conducted for all other Boolean operators. The case of a negated constraint is straightforward: truth and falsity swap their roles. Conjunction is entirely symmetric to disjunction due to De Morgan's law. For example, a query for falsity of the conjunction propagates to both conjuncts while a query for truth need only be propagated to one conjunct. We remark that one can apply such an analysis to other kinds of operators including non-logical ones. Thus, the **cardinality** constraint [HD91] can be handled within this framework.

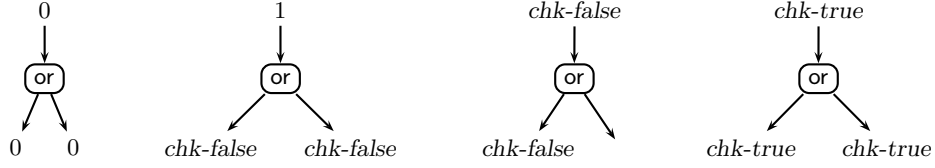


Fig. 1. Control flow through a disjunction

2.1 Controlled Propagation

Irrelevant inference can be prevented by distinguishing whether the truth or the falsity of a constraint matters. This control information arises from truth information and is propagated similarly. By **controlled propagation** we mean constraint propagation that (1) conducts inference according to truth and falsity information and (2) propagates such information.

We now characterise controlled propagation for a complex constraint in decomposition. We are interested in the *effective* propagation, i.e. newly inferred constraints (such as smaller domains) on the original variables rather than on auxiliary Boolean variables. We assume that only individual constraints are propagated¹. This is the usual case in practice.

Theorem 1 (Controlled Propagation). *Controlled and uncontrolled propagation of the constraints of the decomposition of a constraint C are equivalent with respect to the variables of C if only single constraints are propagated.* \square

Proof. Proposition 1 and analogous propositions for the other Boolean operators.

In the following, we explain a formal framework for maintaining and reacting to control information.

Control store. Constraints communicate truth information by shared Boolean variables. Similarly, we think of control information being communicated between constraints by shared sets of control flags. As control flags we consider the truth status queries *chk-true*, *chk-false* and the additional flag *irrelevant* signalling **permanent irrelevance**. In this context, ‘permanently’ refers to subsidiary parts of the search, that is, until the next back-tracking. Note that the temporary absence of truth and falsity queries on a constraint is not the same as its irrelevance. We write

C with \mathcal{FS}

to mean that the constraint C can read and update the sequence of control flag sets \mathcal{FS} . One difference between logic and control information communication is that control flows only one way, from a producer to a consumer.

¹ E.g., path-consistency enforcing propagation considers two constraints at a time.

Propagating control. A set of control flags \mathcal{F} is updated by adding or deleting flags. We abbreviate the adding operation $\mathcal{F} := \mathcal{F} \cup \{f\}$ as $\mathcal{F} \cup = f$. We denote by $\mathcal{F}_1 \rightsquigarrow \mathcal{F}_2$ that from now on permanently changes to the control flags in \mathcal{F}_1 are reflected in corresponding changes to \mathcal{F}_2 ; e.g. an addition of f to \mathcal{F}_1 leads to an addition of f to \mathcal{F}_2 .

We employ rules to specify how control information is attached to the constituents of a decomposed complex constraint, and how it propagates. The rule $A \Rightarrow B$ denotes that the conditions in A , consisting of constraints and associated control information, entail the constraints and the updates of control information specified in B . We use *delete* statements in the conclusion to explicitly remove a constraint from the constraint store once it is solved or became permanently irrelevant.

Relevance. At the core of controlled propagation is the principle that reasoning effort should be made only if it is relevant to do so, that is, if the truth or falsity of the constraint at hand is asserted or queried. We reflect this condition in the predicate

$$\begin{aligned} is_relevant(b, \mathcal{F}) \quad := \quad & b = 1 \quad \text{or} \quad chk_true \in \mathcal{F} \quad \text{or} \\ & b = 0 \quad \text{or} \quad chk_false \in \mathcal{F}. \end{aligned} \quad (is_rel)$$

It applies to constraints in the form $C \equiv b$ with \mathcal{F} . We show later that this principle can be applied to primitive constraints.

2.2 Boolean Constraints

We again focus on disjunctive constraints. The following rule decomposes the constraint $(C_1 \vee C_2) \equiv b$ only if the relevance test is passed. In this case the shared control sets are initialised.

$$\begin{aligned} is_relevant(b, \mathcal{F}) \quad \Rightarrow \quad & or(b, b_1, b_2) \text{ with } \langle \mathcal{F}, \mathcal{F}_1, \mathcal{F}_2 \rangle, \\ & C_1 \equiv b_1 \text{ with } \mathcal{F}_1, \mathcal{F}_1 := \emptyset, \\ & C_2 \equiv b_2 \text{ with } \mathcal{F}_2, \mathcal{F}_2 := \emptyset. \end{aligned} \quad (or_dec)$$

The following rules specify how control information propagates through this disjunctive constraint in accordance with Proposition 1:

$$\begin{aligned} b = 1 \quad & \Rightarrow \quad \mathcal{F}_1 \cup = chk_false, \mathcal{F}_2 \cup = chk_false; \\ b_1 = 0 \quad & \Rightarrow \quad \mathcal{F} \rightsquigarrow \mathcal{F}_2, \text{delete } or(b, b_1, b_2); \\ b_2 = 0 \quad & \Rightarrow \quad \mathcal{F} \rightsquigarrow \mathcal{F}_1, \text{delete } or(b, b_1, b_2); \\ b_1 = 1 \quad & \Rightarrow \quad \mathcal{F}_2 \cup = irrelevant, \text{delete } or(b, b_1, b_2); \\ b_2 = 1 \quad & \Rightarrow \quad \mathcal{F}_1 \cup = irrelevant, \text{delete } or(b, b_1, b_2); \\ chk_false \in \mathcal{F} \quad & \Rightarrow \quad \mathcal{F}_1 \cup = chk_false; \\ chk_true \in \mathcal{F} \quad & \Rightarrow \quad \mathcal{F}_1 \cup = chk_true, \mathcal{F}_2 \cup = chk_true; \\ irrelevant \in \mathcal{F} \quad & \Rightarrow \quad \mathcal{F}_1 \cup = irrelevant, \mathcal{F}_2 \cup = irrelevant, \text{delete } or(b, b_1, b_2). \end{aligned} \quad (or_cf)$$

In rule (or_{cf}), we arbitrarily select the first disjunct to receive *chk-false*. For comparison and completeness, here are the rules propagating truth information:

$$\begin{array}{ll} b_1 = 0 & \Rightarrow b = b_2; \\ b_2 = 0 & \Rightarrow b = b_1; \\ b = 0 & \Rightarrow b_1 = 0, b_2 = 0. \end{array} \qquad \begin{array}{ll} b_1 = 1 & \Rightarrow b = 1; \\ b_2 = 1 & \Rightarrow b = 1; \end{array}$$

Control propagation for the negation constraint $\text{not}(b, b_N)$ with $\langle \mathcal{F}, \mathcal{F}_N \rangle$ is straightforward:

$$\begin{array}{ll} b = 1 \text{ or } b = 0 \text{ or } b_N = 1 \text{ or } b_N = 0 & \Rightarrow \text{delete } \text{not}(b, b_N); \\ \text{chk-false} \in \mathcal{F} & \Rightarrow \mathcal{F}_N \cup = \text{chk-true}; \\ \text{chk-true} \in \mathcal{F} & \Rightarrow \mathcal{F}_N \cup = \text{chk-false}; \\ \text{irrelevant} \in \mathcal{F} & \Rightarrow \mathcal{F}_N \cup = \text{irrelevant}. \end{array}$$

The rules for other Boolean operators are analogous. Note that a move from binary to n -ary conjunctions or disjunctions does not affect the control flow in principle, in the same way that the logic is unaffected.

Both *chk-true* and *chk-false* can be in the control set of a constraint at the same time, as it might be in a both positive and negative context. An example is the condition of an if-then-else. On the other hand, if for instance a constraint is not in a negated context, *chk-false* cannot arise.

2.3 Primitive Constraints

Asserting and querying other primitive constraints can be controlled similarly to Boolean constraints. In particular, the relevance condition (*is_rel*) must be satisfied before inspecting a constraint. We furthermore deal with *irrelevant* $\in \mathcal{F}$ as expected, by not asserting the primitive constraint or by deleting it from the set of currently queried or asserted constraints.

When a query on a primitive constraint is inconclusive, it is re-evaluated whenever useful. This can be when elements from a variable domain are removed or when a bound changes. We rely on the constraint solving environment to signal such changes.

Deciding the truth or the falsity of a constraint in general is an expensive operation that requires the evaluation of every variable domain. A primitive $C(X)$ is guaranteed to be true if and only if $C(X) \subseteq D(X)$ and $C(X)$ is non-empty. C is guaranteed to be false if and only if $C(X) \cap D(X) = \emptyset$, where $X = x_1, \dots, x_n$ and $D(X) = D(x_1) \times \dots \times D(x_n)$. For some primitive constraints we can give complete but simpler evaluation criteria, similarly to indexicals [CD96]; see Tab. 1.

Practical constraint solving systems usually maintain domain bounds explicitly. This makes answering the truth query for equality constraints and the queries for ordering constraints very efficient. Furthermore, the re-evaluation of a query can be better controlled: only changes of the respective bounds are an event that makes a re-evaluation worthwhile.

| Constraint | true if | false if |
|------------|------------------------------------|------------------------------|
| $x \in S$ | $D(x) \subseteq S$ | $D(x) \cap S = \emptyset$ |
| $x = a$ | $ D(x) = 1, D(x) = \{a\}$ | $a \notin D(x)$ |
| $x = y$ | $ D(x) = D(y) = 1, D(x) = D(y)$ | $D(x) \cap D(y) = \emptyset$ |
| $x \leq y$ | $\max(D(x)) \leq \min(D(y))$ | $\min(D(x)) > \max(D(y))$ |

Table 1. Primitive constraint queries (S is a constant set, a is a constant value)

3 Implied Constraints

Appropriate handling of implied constraints fits naturally into the control propagation framework. Suppose the disjunctive constraint $C_1 \vee C_2$ implies C_{\triangleright} ; that is, $(C_1 \vee C_2) \rightarrow C_{\triangleright}$ is always true. Logically, C_{\triangleright} is redundant. In terms of constraint propagation, it may not be, however.

Consider the disjunction $(x = y) \vee (x < y)$, which implies $x \leq y$. Assume the domains are $D(x) = \{4, 5\}$, $D(y) = \{3, 4, 5\}$. Since the individual disjuncts are not false, there is no propagation from the decomposition. In order to conclude $x \leq y$ and thus $D(y) = \{4, 5\}$ we associate the constraint with its implied constraint.

We write a disjunctive constraint annotated with an implied constraint as

$$C_1 \vee C_2 \triangleright C_{\triangleright}.$$

To benefit from the propagation of C_{\triangleright} , we could represent this constraint as $(C_1 \vee C_2) \wedge C_{\triangleright}$. However, this representation has the shortcoming that it leads to redundant propagation in some circumstances. Once one disjunct, say, C_1 , is known to be false, the other disjunct, C_2 , can be imposed. The propagation of C_{\triangleright} is then still executed, however, while it is subsumed by that of C_2 . It is desirable to recognise that C_{\triangleright} is operationally redundant at this point. We capture this situation by enhancing the decomposition rule (or_{dec}) as follows:

$$\begin{aligned}
(C_1 \vee C_2 \triangleright C_{\triangleright}) \equiv b \text{ with } \mathcal{F} &\Rightarrow \text{or}_{\triangleright}(b, b_1, b_2, b_{\triangleright}) \text{ with } \langle \mathcal{F}, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_{\triangleright} \rangle, \\
C_1 \equiv b_1 \text{ with } \mathcal{F}_1, \mathcal{F}_1 &:= \emptyset, \\
C_2 \equiv b_2 \text{ with } \mathcal{F}_2, \mathcal{F}_2 &:= \emptyset, \\
C_{\triangleright} \equiv b_{\triangleright} \text{ with } \mathcal{F}_{\triangleright}, \mathcal{F}_{\triangleright} &:= \emptyset.
\end{aligned}$$

Additionally to the control rules for regular disjunctive constraints shown earlier, we now also use the following four rules:

$$\begin{aligned}
b_{\triangleright} = 0 &\Rightarrow b = 0; & b_1 = 0 &\Rightarrow \mathcal{F}_{\triangleright} \cup = \text{irrelevant}, \text{ delete } \text{or}_{\triangleright}(b, b_1, b_2, b_{\triangleright}); \\
b = 1 &\Rightarrow b_{\triangleright} = 1; & b_2 = 0 &\Rightarrow \mathcal{F}_{\triangleright} \cup = \text{irrelevant}, \text{ delete } \text{or}_{\triangleright}(b, b_1, b_2, b_{\triangleright}).
\end{aligned}$$

We envisage the automated discovery of implied constraints, but for now we assume manual annotation.

4 Subconstraint Sharing: From Trees to DAGs

The straightforward decomposition of complex constraints can contain unnecessary copies of the same subconstraint in different contexts. The dual constraint graph (whose vertices are the constraints and whose edges are the variables) is a tree, while often a directed acyclic graph (DAG) gives a logically equivalent but more compact representation. See, for example, CDDs [CY05].

We can apply controlled propagation to complex constraints represented in DAG form. We need to account for the multiplicity of a constraint when handling queries on it: the set of control flags now becomes a *multiset*, and in effect, we maintain *reference counters* for subconstraints. Control flags need to be properly subtracted from the control set of a constraint. For the sake of a simple example, consider the constraint $(C \vee C_1) \wedge (C \vee C_2)$. Fig. 2 shows a decomposition of it.

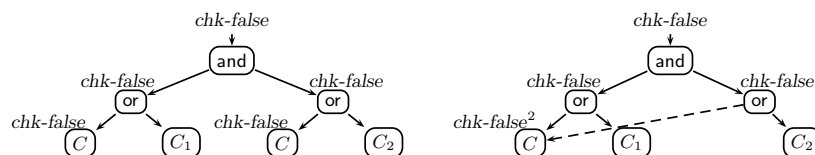


Fig. 2. Left: no sharing. Right: sharing with reference counting

Another example is the condition in an if-then-else constraint. Opportunities for shared structures arise frequently when constraints are defined in terms of subconstraints that in turn are constructed by recursive definitions.

5 Case Studies

We examine several constraints studied in the literature and show that their decomposition benefits from controlled propagation.

Literal Watching. The DPLL procedure for solving the SAT problem uses a combination of search and inference and can be viewed as a special case of constraint programming. Many SAT solvers based on DPLL employ unit propagation with *2-literal watching*, e. g. Chaff [MMZ⁺01]. At any time, only changes to two literals per clause are tracked, and consideration of other literals is postponed.

Let us view a propositional clause as a Boolean constraint. We define

$$\text{clause}(x_1, \dots, x_n) \quad := \quad x_1 = 1 \vee \text{clause}(x_2, \dots, x_n)$$

and show in Fig. 3 the decomposition of $\text{clause}(x_1, \dots, x_n)$ as a graph for controlled and uncontrolled propagation (where $D(x_i) = \{0, 1\}$ for all x_i). Both propagation variants enforce domain-consistency if the primitive equality constraints do and the variables are pairwise different. This corresponds to unit propagation.

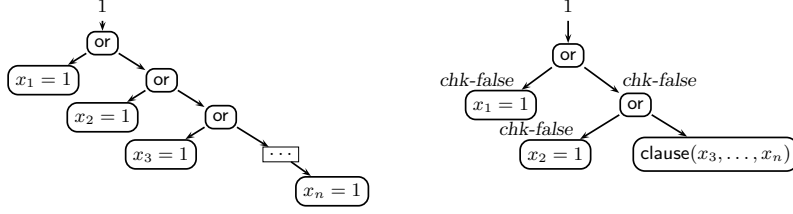


Fig. 3. Uncontrolled versus controlled decomposition of clause

Uncontrolled decomposition expands fully into $n - 1$ Boolean **or** constraints and n primitive constraints $x_i = 1$. Controlled decomposition only expands into two **or** constraints and the first two primitive constraints $x_1 = 1$, $x_2 = 1$. The leaf node marked **clause**(x_3, \dots, x_n) is initially not expanded as neither assertion nor query information is passed to it. The essence is that the first **or** constraint results in two *chk-false* queries to the subordinate **or** constraint which passes this query on to just one disjunct. This structure is maintained with respect to new information such as variable instantiations. No more than two primitive equality constraints are ever queried at a time. A reduction of inference effort as well as of space usage results.

Controlled propagation here corresponds precisely to 2-literal watching.

Disequality of Tuples. Finite domain constraint programming generally focuses on variables over the integers. Sometimes, higher-structured variable types, such as sets of integers, are more appropriate for modelling. Many complex constraints studied in the constraint community are on a sequence of variables and can thus naturally be viewed as constraining a variable whose type is tuple-of-integers. The recent study [QW05] examines how some known constraint propagation algorithms for integer variables can be lifted to higher-structured variables. One of the constraints examined is **alldifferent** on tuples, which requires a sequence of variables of type tuple-of-integers to be pairwise different. Its straightforward definition is

$$\text{alldifferent_tp}(\langle X_1, \dots, X_n \rangle) := \bigwedge_{i,j \in 1, \dots, n, i < j} \text{different_tp}(X_i, X_j),$$

where

$$\text{different_tp}(\langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_m \rangle) := \bigvee_{i \in 1, \dots, m} x_i \neq y_i.$$

Let us examine these constraints with respect to controlled propagation. The **different_tp** constraint is a large disjunction, and it behaves thus like the **clause** constraint studied in the previous section – at most two disjuncts $x_i \neq y_i$ are queried for falsity at any time.

Deciding the falsity of a disequality constraint is particularly efficient when the primitive constraints in Tab. 1 are used, i. e. falsity of disequality when the

domains are singletons. If the domains are not singletons, re-evaluation of the query is only necessary once that is the case. In contrast, a truth query for a disequality is (more) expensive as the domains must be intersected, and, if inconclusive, should be re-evaluated whenever any domain change occurred.

The `alldifferent_tp` constraint is a conjunction of $\binom{n}{2}$ `different_tp` constraints. Therefore, controlled propagation queries at most $n(n-1)$ disequality constraints for falsity at a time. Uncontrolled propagation asserts all $n(n-1)m/2$ reified disequality constraints and in essence queries truth and falsity of each. Using controlled rather than uncontrolled decomposition-based propagation for `alldifferent_tp` saves substantial effort without loss of effective propagation.

We remark that a specialised, stronger but non-trivial propagation algorithm for this case has been studied in [QW05]. The controlled propagation framework is then useful when specialised algorithms are not readily available, for example due to a lack of expertise or resources in the design and implementation of propagation algorithms.

Lexicographic Ordering Constraint. It is often desirable to prevent symmetries in constraint problems. One way is to add symmetry-breaking constraints such as the lexicographic ordering constraint [FHK⁺02]. A straightforward definition is as follows:

$$\begin{aligned} \text{lex}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle) &:= x_1 < y_1 \\ &\vee \\ &x_1 = y_1 \wedge \text{lex}(\langle x_2, \dots, x_n \rangle, \langle y_2, \dots, y_n \rangle) \\ &\vee \\ &n = 0 \end{aligned}$$

With this definition, propagation of the decomposition does not always enforce domain-consistency. Consider $\text{lex}(\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle)$ with the domains $D(x_1) = D(x_2) = D(y_2) = \{3..5\}$ and $D(y_1) = \{0..5\}$. Controlled decomposition results in the reified versions of $x_1 < y_1$, $x_1 = y_1$, $x_2 < y_2$ connected by Boolean constraints. None of these primitive constraints is true or false. Yet we should be able to conclude $x_1 \leq y_1$, hence $D(y_1) = \{3..5\}$, from the definition of lex .

The difficulty is that the naive decomposition is weaker than the logical definition because it only reasons on the individual primitive constraints. However, it is easy to see that $x_1 \leq y_1$ is an implied constraint in the sense of Section 3, and we can annotate the definition of lex accordingly:

$$\begin{aligned} \text{lex}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle) &:= x_1 < y_1 \\ &\vee \\ &x_1 = y_1 \wedge \text{lex}(\langle x_2, \dots, x_n \rangle, \langle y_2, \dots, y_n \rangle) \\ &\supset x_1 \leq y_1 \\ &\vee \\ &n = 0 \end{aligned}$$

We state without proof that propagation of the constraints of the decomposition enforces domain-consistency on lex if the annotated definition is used.

Tab. 2 represents a trace of lex on the example used in [FHK⁺02], showing the lazy decomposing due to controlled propagation. We collapse several atomic inference steps and omit the Boolean constraints, and we write $v_{i..j}$ to abbreviate v_i, \dots, v_j . Observe how the implied constraints $x_i \leq y_i$ are asserted, made irrelevant and then deleted. The derivation ends with no constraints other than $x_3 < y_3$ queried or asserted.

| Asserted | Set of constraints queried for falsity | Variable domains | | | | |
|--|--|---------------------------|---------------------------|------------------------------|------------|------------|
| | | x_1 | x_2 | x_3 | x_4 | x_5 |
| | | y_1 | y_2 | y_3 | x_4 | y_5 |
| $\text{lex}(\langle x_{1..5} \rangle, \langle y_{1..5} \rangle)$ | | $\{2\}$ | $\{1, 3, 4\}$ | $\{1..5\}$ | $\{1..2\}$ | $\{3..5\}$ |
| $x_1 \leq y_1$ | $x_1 < y_1, x_1 = y_1, x_2 < y_2$ | $\{0..2\}$ | $\{1\}$ | $\{0..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| $x_2 \leq y_2$ | $x_2 < y_2, x_2 = y_2, x_3 < y_3$ | $\{2\}$ | $\{1, 3, 4\}$ | $\{1..5\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{0..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| $x_3 \leq y_3$ | $x_3 < y_3, x_3 = y_3, x_4 < y_4$ | $\{2\}$ | $\{1\}$ | $\{1..5\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{0..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| $x_3 \leq y_3$ | $x_3 < y_3, x_3 = y_3, x_4 = y_4, x_5 < y_5$ | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| $x_3 \leq y_3$ | $x_3 < y_3, x_3 = y_3, x_4 = y_4, x_5 = y_5$ | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| $x_3 < y_3$ | | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{1..4\}$ | $\{0..1\}$ | $\{0..2\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{1..3\}$ | $\{1..2\}$ | $\{3..5\}$ |
| | | $\{2\}$ | $\{1\}$ | $\{2..4\}$ | $\{0..1\}$ | $\{0..2\}$ |

Table 2. An example of controlled propagation of the lex constraint

6 Implementation and Benchmarks

We implemented a prototype of the controlled propagation framework in the CLP system ECLⁱPS^e [WNS97], using its predicate suspension features and attributed variables to handle control information. The implementation provides controlled propagation for the basic Boolean and primitive constraints, and it handles implied constraints. Structure-sharing by a DAG-structured decomposition is not supported.

We conducted several simple benchmarks to compare controlled and uncontrolled propagation on constraint decompositions, using the `clause`, `different_tp`, `alldifferent_tp` and `lex` constraints. A benchmark consisted of finding a solution

| | clause | | | | different_tp | | | | alldifferent_tp | | | | lex | | | |
|------------------|--------|----|----|----|--------------|----|----|----|-----------------|----|----|----|-----|----|----|----|
| nb. of variables | 5 | 10 | 20 | 50 | 5 | 10 | 20 | 50 | 5 | 10 | 20 | 50 | 5 | 10 | 20 | 50 |
| runtime (%) | 100 | 69 | 50 | 38 | 88 | 84 | 67 | 62 | 66 | 38 | 23 | 11 | 138 | 92 | 69 | 54 |

Table 3. Benchmark results: controlled propagation (uncontrolled prop. = 100%)

to a single constraint. For the uncontrolled propagation benchmark, the constraint was simply decomposed into built-in Boolean and primitive constraints of ECLⁱPS^e, and implied constraints (in *lex*) were conjunctively added to their respective premise.

The number of variables in the respective tuple(s) was varied between five and 50. For the *alldifferent_tp* benchmark, we chose 20 tuples. The variables ranged over the interval $\{1..10\}$ (except for *clause*). Solutions to the constraints were searched by randomly selecting a variable and a value in its domain. This value was either assigned or excluded from its domain; this choice was also random. To obtain meaningful averages, every individual solution search was run a sufficient number of times (typically a few 10000) so that the total computation time was roughly 15 s. Each of these runs used a new initialisation of the pseudo-random number generator resulting in a possibly different solution, while the benchmark versions (controlled vs. uncontrolled propagation) used the same initial value to obtain identical search trees. Every experiment was repeated five times. In Tab. 3, we give the relative solving time with controlled propagation, based on the corresponding uncontrolled propagation benchmark taken to be 100%.

The benchmarks show that controlling propagation can reduce the propagation time. The reduction is especially substantial for high-arity constraints. For low-arity constraints, the extra cost of maintaining control information in our implementation can outweigh the saving due to less propagation. While we have not measured the space usage of the two propagation approaches, it follows from the analyses in Section 5 that using controlled propagation for the considered constraints often also requires less space, since constraints are decomposed only when required.

We remark that efficiency was a minor concern in our high-level, proof-of-concept implementation; consequently we expect that it can be improved considerably. For example, for constraints that are in negation normal form (all constraints in our benchmark), the control flag *chk-true* is never created. A simpler subset of the control propagation rules can then be used.

7 Final Remarks

Related Work. In terms of foundations, the controlled propagation framework can be described as a refined instance of the CLP scheme (see [JM94]), by a subdivision of the set of active constraints according to their associated truth and falsity queries. Concurrent constraint programming (CCP) [Sar93], based on asserting and querying constraints, is closely related; our propagation framework

can be viewed as an extension in which control is explicitly addressed and dealt with in a fine-grained way. A practical CCP-based language such as CHR [Frü98] would lend itself well to an implementation. For example, control propagation rules with delete statements can be implemented as simplification rules.

A number of approaches address the issue of propagation of complex constraints. The proposal of [BW05] is to view a constraint as an expression from which sets of inconsistent or valid variable assignments (in extension) can be computed. It focuses more on the complexity issues of achieving certain kinds of local consistencies. The work [BCP04] studies semi-automatic construction of propagation mechanisms for constraints defined by extended finite automata. An automaton is captured by signature (automaton input) constraints and state transition constraints. Signature constraints represent groups of reified primitive constraints and are considered pre-defined. They communicate with state transition constraints via constrained variables, which correspond to tuples of Boolean variables of the reified constraints in the signature constraints. Similarly to propagating the constraint in decomposition, all automata constraints are propagated independently of each other.

Controlled propagation is similar to techniques used in NOCLAUSE, a SAT solver for propositional non-CNF formulas [TBW04], which in turn lifts techniques such as 2-literal watching from CNF to non-CNF solvers. We describe here these techniques in a formal, abstract framework and integrate non-Boolean primitive constraints and implied constraints, thus making them usable for constraint propagation.

Conclusion. We have proposed a new framework for propagating arbitrary complex constraints. It is characterised by viewing logic and control as separate concerns. We have shown that the controlled propagation framework explains and generalises some of the principles on which efficient manually devised propagation algorithms for complex constraints are based. By discussing an implementation and benchmarks, we have demonstrated feasibility and efficiency. The practical benefits of the controlled propagation framework are that it provides *automatic* constraint propagation for *arbitrary* logical combinations of primitive constraints. Depending on the constraint, controlling the propagation can result in substantially reduced usage of time as well as space.

Our focus in this paper has been on reducing unnecessary inference steps. The complementary task of automatically identifying and enabling useful inference steps in our framework deserves to be addressed. It would be interesting to investigate if automatic reasoning methods can be used to strengthen constraint definitions, for instance by automatically deriving implied constraints.

Acknowledgements

We thank the anonymous reviewers for their comments. This paper was written while Roland Yap was visiting the Swedish Institute of Computer Science and

their support and hospitality are gratefully acknowledged. The research here is supported by a NUS ARF grant.

References

- BCP04. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In Wallace [Wal04], pages 107–122.
- BW05. F. Bacchus and T. Walsh. Propagating logical combinations of constraints. In L. P. Kaelbling and A. Saffioti, editors, *Proc. of International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 35–40, 2005.
- CCLW99. B. M. W. Cheng, K. M. F. Choi, J. H.-M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: An experience report. *Constraints*, 4(2):167–192, 1999.
- CD96. P. Codognot and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
- CY05. K. C. K. Cheng and R. H. C. Yap. Constrained decision diagrams. In M. M. Veloso and S. Kambhampati, editors, *Proc. of 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 366–371. AAAI Press, 2005.
- FHK⁺02. A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. Van Hentenryck, editor, *Proc. of 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 93–108. Springer, 2002.
- Fre82. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
- Frü98. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
- HD91. P. Van Hentenryck and Y. Deville. The Cardinality operator: A new logical connective for constraint logic programming. In K. Furukawa, editor, *Proc. of 8th International Conference on Logic Programming (ICLP'91)*, pages 745–759. MIT Press, 1991.
- JM94. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 & 20:503–582, 1994.
- Kow79. R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
- MMZ⁺01. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of 38th Design Automation Conference (DAC'01)*, 2001.
- QW05. C.-G. Quimper and T. Walsh. Beyond finite domains: The All Different and Global Cardinality constraints. In P. van Beek, editor, *Proc. of 11th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3709 of *LNCS*, pages 812–816. Springer, 2005.
- Rég94. J.-C. Régin. A filtering algorithm for constraints of difference in csps. In *Proc. of 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 362–367. AAAI Press, 1994.
- Sar93. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
- TBW04. Chr. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In Wallace [Wal04], pages 663–678.
- Wal04. M. Wallace, editor. *Proc. of 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*. Springer, 2004.

- WNS97. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.